

DOCUMENT RESUME

ED 388 254

IR 017 403

AUTHOR Heift, T.; McFetridge, P.
TITLE The Intelligent Workbook.
PUB DATE 94
NOTE 7p.; In: Educational Multimedia and Hypermedia, 1994. Proceedings of ED-MEDIA 94--World Conference on Educational Multimedia and Hypermedia (Vancouver, British Columbia, Canada, June 25-30, 1994); see IR 017 359.
PUB TYPE Reports - Descriptive (141) -- Speeches/Conference Papers (150)
EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Authoring Aids (Programming); Computer Assisted Instruction; Computer System Design; Feedback; Foreign Countries; German; *Intelligent Tutoring Systems; *Second Language Instruction; *Workbooks

ABSTRACT

An authoring system is defined as a computer program which eliminates the need for course instructors to learn computer programming, but instead constructs pedagogical software based on instructors' responses to questions on teaching strategy and course material. This paper outlines an authoring system which not only exhibits this ease of use and flexibility in its design, but demonstrates intelligent behavior, one of the significant interactive qualities of computer assisted language learning. Intelligence is simulated by daemons (a program submodule, typically highly parameterized), each of which seeks a specific error in student input, providing the student with immediate error-contingent feedback. The user defines a daemon by providing a name and specifying the action the daemon will perform. The actions correspond to three decision points: string comparison, lexical search, and sentence search. The model, which allows the course designer to create any type of language exercise, is demonstrated in German as the target language, and English as the source language. An intelligent workbook is constructed through selecting, from a pool of exercises that has been created, those that will appear in the workbook, and assigning an order to them. The system described here was written in Allegro CommonLisp(TM) and runs on the Apple Macintosh(TM). Contains five references. (Author/MAS)

* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *

- ☐ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS
MATERIAL HAS BEEN GRANTED BY

Gary H. Marks

TO THE EDUCATIONAL RESOURCES
INFORMATION CENTER (ERIC)."

The Intelligent Workbook

T. HEIFT and P. MCFETRIDGE

Department of Linguistics

Simon Fraser University, Burnaby, B.C. Canada V5A 1S6

E-mail: heift@sfu.ca mcfet@cs.sfu.ca

Abstract: Burghardt (1984) defines an authoring system as a computer program which eliminates the need for course instructors to learn computer programming but instead constructs pedagogical software based on the instructors responses to questions on teaching strategy and course material. In this paper we will outline an authoring system which not only exhibits this ease of use and flexibility in its design, but demonstrates intelligent behavior, one of the significant interactive qualities of Computer-Assisted Language Learning. Intelligence is simulated by daemons each of which seeks a specific error in student input, providing the student with immediate error-contingent feedback. The model, which allows the course designer to create any type of language exercise is demonstrated with German as the target language and English as the source language.

Introduction

All compelling arguments for Computer-Assisted Language Learning (CALL) assume intelligent responses to students' input. Without intelligence the computer is merely another medium for presenting information, one not especially preferable to a static medium such as print. In order to go beyond the multiple choice questions, relatively uninformative answer keys and gross mainstreaming of students characteristic of workbooks, a model should emulate significant aspects of a student-teacher interaction.

This paper presents an authoring system for CALL. The system is designed to produce an intelligent tutor, rather than an electronic workbook: the principal advantage and contradistinction being that an intelligent tutoring system provides the student with evaluative or error-contingent feedback (Alessi and Trollip, 1985). Venezky & Osin (1991) stress that "for almost all cognitive learning, instruction is enhanced by evaluative feedback. In many cases it is essential, if any learning is to occur. Translation of a foreign language is a prime example of the latter situation (pg. 9)."

Turning to a concrete example of error-contingent feedback, consider the following grammar exercise intended to practice word order in subordinate clauses in German. In the main clause the verb occurs in second position; however in a subordinate clause the verb must be in sentence-final position. The student is presented with the exercise:

- (1) Build a sentence with the words provided:

Ich / nehmen / zwei / Menüs / weil / ich / haben / Hunger.

The simplest system merely reports an error if the student does not supply the correct answer, "Ich nehme zwei Menüs, weil ich Hunger **habe**", perhaps revealing the correct answer after a number of tries. By comparison, error-contingent feedback responds with a description of the error, and performs a deeper linguistic analysis in order to isolate the source of the error. If the student responds with "Ich nehme zwei Menüs, weil ich ***habe** Hunger, the system responds with "incorrect word order in subordinate clause". The more detailed error analysis guides the student toward the correct answer, and provides an invaluable record of the student's performance for later evaluation and remedial work.

The error-contingent feedback which serves as the basis for effective teaching in the language classroom can be transferred to the computer. This has been previously demonstrated with an intelligent tutoring system on a small micro computer platform teaching subordinate clauses in German (Heift 1993, Heift & McFetridge 1993). In this project, all error checking routines — which we have called daemons — were built rather painstakingly. This process is time consuming and requires expert knowledge of programming as well as second language

instruction. The project we report here is an authoring system for creating intelligent workbooks. It provides a method for specifying which daemons are required for each exercise and how they are parameterized. The computer code which analyses student errors is automatically generated.

The Daemon Approach

Error-contingent feedback is provided through the use of daemons which only seek errors relevant to the exercise. A daemon is a program submodule, typically highly parameterized, which seeks a particular error and takes remedial action when that error is discovered. Daemons are conceptually simple but can simulate intelligence. They achieve this, in part, because the pedagogical principles of 'selection' and 'gradation' constrain the problem domain—the range of possible errors in any given exercise is small, or at least finite, compared to many natural language processing tasks. They are simple in the sense that each daemon is responsible for checking only one type of error—in our example above, for instance, one daemon would test solely for word order, while separate daemons would each check verb inflection, spelling, etc. This modularity allows the programmer to create a pool of daemons for each exercise, adding daemons from previous exercises to the current one. The daemon approach ensures relevance of response—the order in which daemons are activated keeps the point of the current exercise salient. Additionally, the overall system can easily be extended to encompass new phenomena by adding new daemons to the pool.

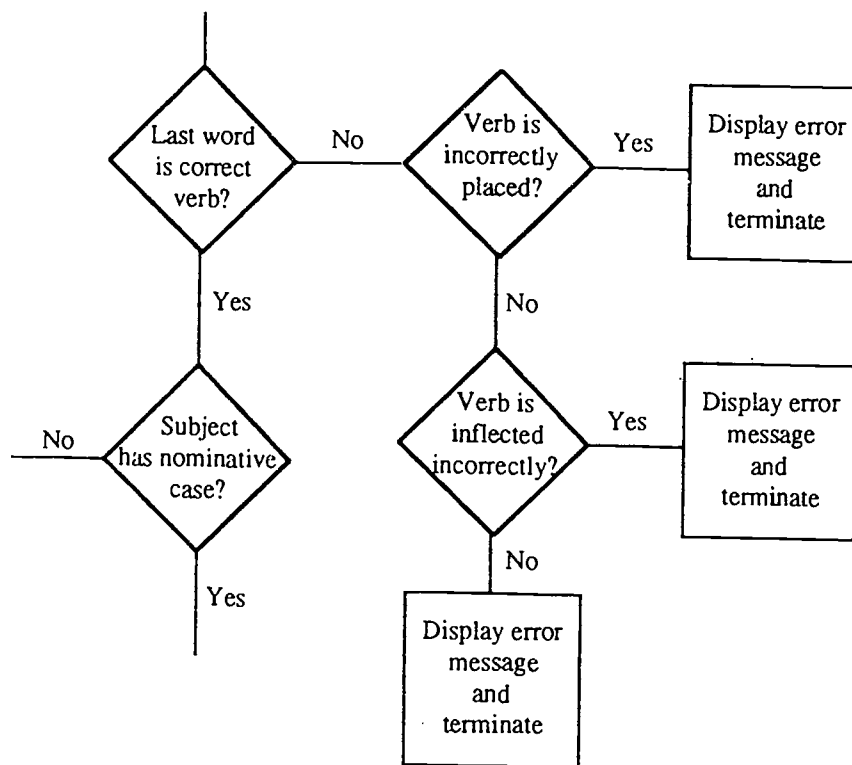
The daemon approach also allows us to make efficient use of a small computer platform, which in turn improves the student/program interaction. Generally, the amount of knowledge and processing time required by a tutoring program increases with the sophistication of the error analysis. We can, however, establish an approximate hierarchy which reflects the kinds of errors most likely to occur, and call the daemons in that order. The hierarchy itself is based on what has been taught most recently, and to some extent, on a contrastive analysis between the native/target language of the student. An additional consideration might be the complexity of the grammatical construction. We can reasonably assume that given two constructs of different complexity, the more difficult of the two will result in more frequent and more persistent errors.

Conceptually powerful and computationally efficient, the daemon approach offers still another attraction: reusability. In the same way that daemons can be conjoined and brought to bear on a series of exercises, much of the research and development time spent in creating one tutoring program can be applied to the next. The rationale underlying an intelligent tutor's design then provides the framework for a higher-level authoring system.

Our goal is to produce tutoring systems which provide intelligent and informative feedback to students' errors. Ours is a more ambitious project than many on two parameters. First, we are attempting to open up the domain of errors which the system can analyze. The simplest system associates messages with particular responses and constrains the student to select one. We wish to allow students freedom in their responses and consequently require greater generality in the routines which analyze the students's input. Second, the system we present here is not a tutoring system itself; it is a workbench for creating a tutoring system. We have decomposed the process of error detection into a small set of subroutines which the program writer can use to create daemons and to build exercises. The resulting tutoring system is one we call an *intelligent workbook* because it has the superficial appearance of an electronic workbook, but incorporates sophisticated error analysis and feedback.

We illustrate how a set of daemons is constructed by first considering the problem of checking that the verb in the example above is inflected correctly and is in the correct position. The portion of a flow chart for an exercise relevant to this task is given in figure 1. Each decision point in the flow chart corresponds to a daemon. The first daemon checks that the final word in the sentence is the correctly inflected verb. It functions as a gateway to other daemons: if the last word in the sentence is not the expected word, the flow of control is passed to daemons which determine what error has been made. The first of these scans the sentence for the verb and if successful displays an error message informing the student that the verb is not in the correct position. It terminates the program to give the student an opportunity to correct the mistake. The second daemon scans the dictionary entry of the verb for an inflected form corresponding to that which the student entered. If it finds one, then the verb must be incorrectly inflected and an error message reporting this is displayed. Finally, if the student's error is so severe that neither daemon can successfully analyze it, a generic error message is displayed.

Figure 1



A System Overview

The task of the program writer is to convert the flow chart in figure 1 into a program. The system we describe here is a workbench which modularizes the construction of a program and presents the writer with a set of packaged daemons. Creating an intelligent workbook is a process of several well-defined stages.

In the first stage, the set of daemons that will be required is defined. We describe this process in the section on daemon functionality. We view this process as iterative and cumulative, and consequently have provided editors which allow the program writer to amend the definitions of daemons. It is also possible to add new daemons to the pool at any time. As a consequence, it is possible to begin with a workbook which can respond to only the grossest errors. Over time, the program writer can build greater functionality into the workbook without changing its overall structure.

After the pool has been defined, the writer defines the exercises which comprise the workbook. As in the print medium, an exercise has a set of instructions and requests input from the student. Part of the definition of an exercise is the set of daemons which are assigned to the exercise. When a daemon is assigned to an exercise, its *parameters* must also be assigned. For example, the task given to a student in example (1) above requires that the verb be placed at the end of the sentence. The daemon that first checks that the last word in the student's answer is the verb has two parameters: the correct answer and the position of the word in the student's answer which must correspond to the correct answer.

The final stage in building the workbook is defining the order in which the exercises are to appear and saving the workbook. The workbook is a stand alone application which presents the student with a series of exercises and applies the daemons to the students responses to each exercise.

Daemon Functionality

The flow chart in figure 1 demonstrates three kinds of error analyses. The first decision point compares two forms and takes action based on the success or failure of the comparison. The second decision point relies on a scan of the sentence. If the verb can be found somewhere in the sentence, the associated error message is dis-

played. The final decision point searches the lexicon for the form which the student entered. To this point of our analysis of error detection, this list exhausts the possibilities.

As an example, the definition of the daemon corresponding to the second decision point is illustrated in figure 2.

Figure 2

Daemon Builder

Name

Type
☒ Positive
☐ Negative

Action
☐ String Comparison
☐ Lexical Search
☒ Sentence Search

☒ Terminate

☒ User Model

☒ Message:
The verb is in the wrong position.

The user defines a daemon by providing a name and specifying the action which the daemon will perform. The actions correspond to three decision points in the example in figure 1:

(1) String Comparison

In a substring search daemons are directed to specific positions within the sentence. This applies to daemons which check word position or verb inflection, as in example (1). Further examples are daemons which check prepositions, inflection of attributive adjectives, gender, case, etc.

(2) Lexical Search

In a lexical search the daemons refer to a database, such as a glossary, or dictionary. For example, in the flow chart above the third decision point corresponds to an attempt to find the last word in the sentence in the dictionary under the lexical entry of **haben**. If it is found, then the student has placed the verb in the correct position but has inflected it incorrectly.

(3) Sentence Search

This kind of daemon searches the entire sentence for specific strings; for example, if a verb is not found in the correct position, scanning the entire sentence for the verb will establish whether the student has placed the verb incorrectly. The example in figure 2 illustrates the construction of a daemon of this type. As the flow chart in figure 1 illustrates, this daemon is activated only if it is previously established that the last string in the sentence is not a correctly inflected form of the verb. The ScanForVerb daemon will look through the sentence for the verb and display an error message if it successfully find the verb.

The *type* distinction is equivalent to the *yes/no* paths in the flow chart. A positive type will display an associated message if the specified action is successful. A negative type will display an associated message if the specified action is unsuccessful. In the flow chart in figure 1, the daemon corresponding to the first decision point will be negative. Its function is not to display a message but to serve as a gateway to the other daemons.

Two other properties of a daemon are determined by the state of the check boxes in the lower left. If the *terminate* box is checked, all further processing is stopped after the message is displayed. This feature gives the student an opportunity to correct the error before presentation of other errors. By default, all daemons which display a message terminate.

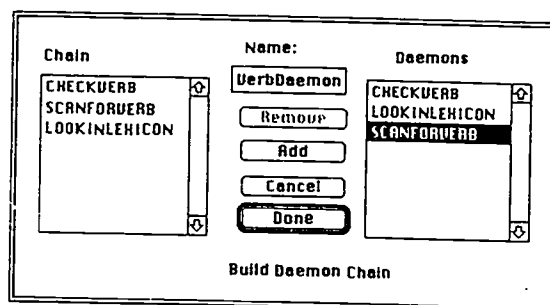
The *user model* check box establishes a count for the error tracked by the daemon. Each time a daemon successfully discovers an error, the count associated with it is incremented. This count can be displayed at the end of a session or be used to direct the student to remedial exercises.

Interrelation of Daemons

It is often necessary for daemons to cooperate in the analysis of a student's input. It is also useful to be able to refer to a large group of daemons by a single name; this is particularly true if the same set of daemons is used in a number of exercises. We have provided two different ways of combining daemons into larger program structures.

The simplest method is to simply group daemons into an ordered set. This set is named and can be applied to an exercise by name. It is also possible for daemons to be mutually exclusive. This situation arises when an error has been discovered, but there are many possible sources of the error. If a daemon successfully fires, it is not necessary to check the others. In figure 3, a chain of daemons with this latter property is constructed. The name of this daemon is VerbDaemon. It consists of three daemons corresponding to the three decision points in the flow chart in figure 1. The first checks that the verb is correct. If it is, the daemon signals success and the rest of the daemons in the chain are ignored. If the verb is incorrect, the other daemons are applied. The ScanForVerb daemon was defined in figure 2. The LookInLexicon daemon corresponds to the third decision point in the flow chart in figure 1.

Figure 3

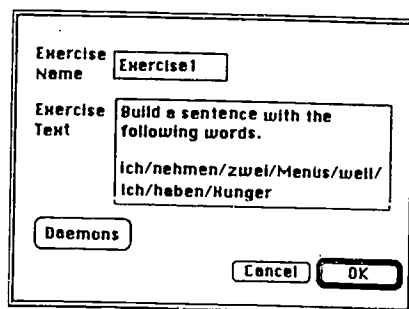


Groups of daemons which are ordered but not mutually exclusive can be built up similarly.

Defining an Exercise

The penultimate step in creating a workbook is setting the exercises. We provide a window in which the text of each exercise is specified. The exercise from the example in (1) above is defined in figure 4. The daemon VerbDaemon, defined in figure 3, is assigned to this exercise. At this point the system requests values for the parameters of each of the subdaemons. The CheckVerb daemon will be parameterized with *habe* and 8. That is, the answer is incorrect if the 8th word is not *habe*. The ScanForVerb daemon will be parameterized with *habe*. This daemon will scan the student's input for this exercise and signal the associated error message if *habe* is found. Finally, the LookInLexicon daemon will be parameterized with *haben* and 8. It will search the lexical entry of *haben* for the word found in 8th position of the input and signal the associated error message if it is found.

Figure 4

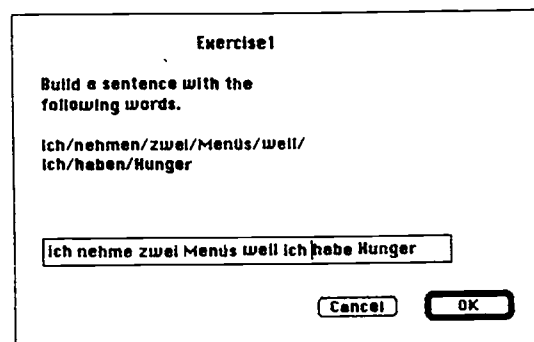


As mentioned previously, as the pool of daemons is enlarged new functionality can be added to an exercise without necessarily perturbing the daemons which have been previously assigned to it. A workbook which includes the exercise in figure 4 may be first produced with the simple set of daemons which we have used in illustration but is not restricted to them. As the program writer becomes familiar with the range of errors which students make in this exercise, the set of daemons can be enriched to accommodate them and added to the exercise.

Building an Intelligent Workbook

The final step in the process of creating a workbook is selecting from the pool of exercises that has been created, those which will appear in the workbook and assigning an order to them. Once this has been done, the workbook is saved as a stand alone application. The workbook displayed to the student consists of a series of windows in which the exercises are presented and the student's input is analysed. An example of a window is given in figure 5.

Figure 5



Conclusion

The work described here is research in progress. We have initially approached the problem of error analysis as one of anticipating possible errors. We believe that this approach has considerable potential, particularly for exercises where students' errors are easily targeted. We have demonstrated that a tutoring system of considerable sophistication can be built from primitive building blocks.

We intend to add further functionality to the lists of actions which a daemon may take and particularly to include natural language processing as part of the error analysis. Natural language processing is particularly good at analyzing grammatical sentences, but has proved difficult to use in error analysis. Parsers normally fail numerous times in the analysis of a sentence and this natural failure is difficult to separate from real errors. Our initial investigations suggest that by assigning natural language processing to a daemon, it is possible to constrain the search space so that the natural language processing system is attempting to solve a particular problem and its success or failure is relevant to the daemon. Our first efforts will include morphological analysis so that students' treatment of inflections can be more deeply analysed. The system described here was written in Allegro CommonLisp™ and runs on the Apple Macintosh™.

References

- Alessi, S. M. & Trollip, S. R. (1985). *Computer-Based Instruction: Methods and Development*. Englewood: Prentice-Hall, Inc.
- Burghardt, W. (1984). Language Authoring with COMET. *Computers and the Humanities*, 18, 165-172.
- Heift, T. (1993). *Error-Adaptive Computer-Assisted Language Learning for German*. M.A. Thesis, Simon Fraser University.
- Heift, T. & McFetridge P. (1993) Teaching Subordinate Clauses in German. In *Proceedings of the CCALL2*, University of Victoria, British Columbia, Canada.
- Venezky, R. & Osin, L. (1991). *The Intelligent Design of Computer Assisted Instruction*. New York: Longman.

Acknowledgment

This work was supported by a Fellowship from the Advanced Systems Institute of British Columbia.